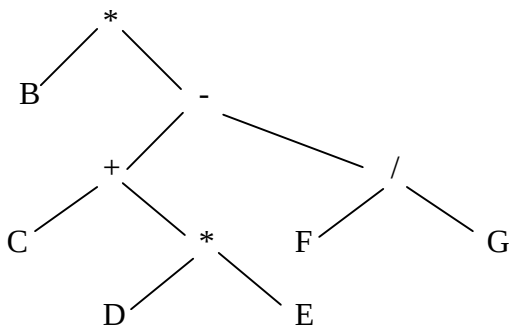
Instruction Architecture:

Consider the (fully parenthesized) arithmetic expression and assignment

A ← (B * ((C + (D*E))– (F/G)))

The arithmetic expression can be represented as a tree by:



which has the postorder traversal

B C D E * + F G / - *

Otherwise known as Polish postfix notation for the expression (note the absence of parentheses
in representing arithmetic expressions this way).

In computing the expression operations requiring three, two, one, or zero operands can be employed. The typical number of operands required by machine language instructions for a CPU determine if it is called a 3-address, 2-address, 1-address, or 0-address architecture.

In a 3-address architecture the 3 operands typically specify left operand, right operand, and target. 3-address format usually results in the shortest sequence of statements to compute and arithmetic expression (at the expense of having to deal with the additional operands in the microcode and the high level language translation). For the above expression a 3-address rendition is:

MPY	D, E, A	[*multiply* D * E *and store the result in* A]
DIV	F, G, T1	[*divide* F / G *and store the result in* T1, *temporary storage*]
ADD	C, A, A	[*add* C + A *and store the result in* A]
SUB	A, T1, A	[*subtract* A – T1 *and store the result in* A]
MPY	B, A, A	[*multiply* B * A *and store the result in* A]

Notice that temporary storage is required for intermediate results, and that the target of the final assignment (A) is used as temporary storage in intermediate steps.

A 2-address architecture follows a convention that one of the operands receives the result of operating on the two operands; e.g., MPY X, Y results in X * Y stored in Y. This tactic results in a longer program that with 3-address, but is simpler to translate. A 2-address rendition for the expression is:

MOVE	E, A	[*prime the target* A *by storing* E *in it*]
MPY	D, A	[*multiply* D*A *storing the result in* A]
MOVE	G, T1	[*prime the temporary* T1 *by storing* G *in it*]
DIV	F, T1	[*divide* F/T1 *storing the result in* T1]
ADD	C, A	[*add* C+A *storing the result in* A]
SUB	A, T1	[*subtract* A-T1 *storing the result in* T1]
MPY	B, T1	[*multiply* B*T1 *storing the result in* T1]
MOVE	T1, A	[*set* A *by storing* T1 *in* A]

This is not the only way to accomplish the task, which is one of the translation complications. Note that the number of temporary storage locations needed is the same as for 3-address.

For a 1-address architecture, an implicit storage location (a CPU *register*, typically) is employed that must be loaded, operated on via the one operand, and then unloaded (if necessary) to capture the intermediate calculation as part of the sequence of instructions that accomplish the desired computation. This typically requires a yet longer program. A 1-address rendition for the expression is:

LOAD	D	[*load* D *into the register*]
MPY	E	[*multiply the register by* E, *result in the register*]
STORE	A	[*capture the result (D*E) by storing the register in* A]
LOAD	F	[*load* F *into the register*]
DIV	G	[*divide the register by* G, *result in the register*]
STORE	T1	[*capture the result (F/G) by storing the register in* T1]
LOAD	C	[*load* C *into the register*]
ADD	A	[*add the register to* A *(which is D*E), result in the register*]
SUB	T1	[*subtract* T1 *from the register, result in the register*]
STORE	A	[*capture the result ((C+(D*E))-(F/G)) by storing the register in* A]
LOAD	B	[*load* B *into the register*]
MPY	A	[*multiply the register by* A, *result in the register*]
STORE	A	[*capture the (final) result by storing the register in* A]

0-address at first blush might appear to be the most complex, but an appropriate instruction sequence can be derived immediately from the Polish postfix form of the statement:

B C D E * + F G / - *

A 0-address rendition utilizes the standard PUSH and POP operations of stacks, which are the LIFO (last in, first out) linear lists used in many software applications. PUSH X simply inserts X on top of the stack (X is *last in*). The reverse operation, POP Y simply removes the top of the stack into Y (Y is *first out*). When a binary operation is performed, the top two elements of the stack are operated on and the result replaces them as the new top of the stack. For the expression under consideration, a 0-address rendition is given by:

```
PUSH    B
PUSH    C
PUSH    D
PUSH    E
MPY             [D*E now the stack top in place of D and E]
ADD             [C+(D*E) is now the stack top]
PUSH    F
PUSH    G
DIV             [F/G is now the stack top]
SUB             [(C+(D*E))-(F/G) is now the stack top]
MPY             [B*((C+(D*E))-(F/G)) is now the stack top]
POP     A
```

Note the slavish adherence to the Polish postfix form. Also note that no temporary storage is utilized!